electronics

MDPI

Article

Efficient Implementation on Low-Cost SoC-FPGAs of TLSv1.2 Protocol with ECC_AES Support for Secure IoT Coordinators

Ahmed Mohamed Bellemou [1,2], Antonio García [3], Encarnación Castillo [3], Nadjia Benblidia [2], Mohamed Anane [4], José Antonio Álvarez-Bermejo [5] and Luis Parrilla [3,*]

[1] Department of System and Multimedia Architecture, Centre de Développement des Technologies Avancées, Baba Hassen, Algiers 16081, Algeria; abellemou@cdta.dz
[2] LRDSI Laboratory, Department of Electronics, Blida 1 University, Blida 09000, Algeria; benblidia@yahoo.com
[3] Departamento Electrónica y Tecnología de Computadores, Universidad de Granada, 18071 Granada, Spain; grios@ugr.es (A.G.); encas@ugr.es (E.C.)
[4] Ecole Supérieure d'Informatique, El Harrach, Algiers 16270, Algeria; m_anane@esi.dz
[5] Departamento Informática, Universidad de Almería, 04120 Almería, Spain; jaberme@ual.es
* Correspondence: lparrilla@ditec.ugr.es; Tel.: +34-958-244-082

Received: 7 October 2019; Accepted: 26 October 2019; Published: 30 October 2019

Abstract: Security management for IoT applications is a critical research field, especially when taking into account the performance variation over the very different IoT devices. In this paper, we present high-performance client/server coordinators on low-cost SoC-FPGA devices for secure IoT data collection. Security is ensured by using the Transport Layer Security (TLS) protocol based on the TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 cipher suite. The hardware architecture of the proposed coordinators is based on SW/HW co-design, implementing within the hardware accelerator core Elliptic Curve Scalar Multiplication (ECSM), which is the core operation of Elliptic Curve Cryptosystems (ECC). Meanwhile, the control of the overall TLS scheme is performed in software by an ARM Cortex-A9 microprocessor. In fact, the implementation of the ECC accelerator core around an ARM microprocessor allows not only the improvement of ECSM execution but also the performance enhancement of the overall cryptosystem. The integration of the ARM processor enables to exploit the possibility of embedded Linux features for high system flexibility. As a result, the proposed ECC accelerator requires limited area, with only 3395 LUTs on the Zynq device used to perform high-speed, 233-bit ECSMs in 413 µs, with a 50 MHz clock. Moreover, the generation of a 384-bit TLS handshake secret key between client and server coordinators requires 67.5 ms on a low cost Zynq 7Z007S device.

Keywords: TLS; ECC; AES; FPGA; Embedded Linux

1. Introduction

The growth in the penetration of the Internet of Things (IoT) [1] in our daily life, be it in fields such as smart homes, smart enterprises, smart hospitals or smart cities, which require a large number of interconnected IoT devices, open the subject of IoT data security concerns. In fact, large amounts of information are transferred through heterogeneous networks, ranging from local wireless sensor networks (WSN) to Wide Area Networks (WAN). Fortunately, Transport Layer Security (TLS) [2] provides an end-to-end network secure information transfer over insecure channels by combining heterogeneous cryptographic protocols like symmetric schemes (e.g., 3DES, AES) [3,4], secure hash functions (e.g., SHA-1, SHA-2, SHA-3) [5,6] and public-key algorithms (e.g., RSA, ECDH, ECDSA) [7]. These last cryptosystems are computationally intensive, due to the complex operations required by public key protocols, and may not be of generalized use due to hardware limitations. However, in controlled environments, such as local WSNs, simplified protocols implemented over compact cryptoprocessors can be a solution [8]. In the case of IoT coordinator nodes, which require transferring data to the Internet, Secure Sockets Layer (SSL) [9] or TLS are the preferred solution.

In the IoT paradigm there are different agents implied, such as sensors, cameras, actuators or microchips, which collect and transfer information through the Internet. As it is difficult to regulate the performance of all IoT devices, security management for IoT applications becomes much more difficult than for a single device [8]. Due to the low performance hardware resources of a large number of IoT agents [9], the targeted cryptographic algorithms are not suitable to be implemented on every IoT device [10]. Hence, we propose to design high-performance client/server coordinators on low-cost SoC-FPGA devices for secure IoT data collection, as shown in Figure 1. The IoT Client coordinators (IoTC1, IoTC2) collect data from IoT agents (A1, A2, A3, A4, A5, A6) and send it to the server through the Internet. The IoT Server coordinator (IoTS) acts as an interface between the IoTCs and the server's memory, where these data will be stored. The secure data transfer between IoTCs and the IoTS is ensured by the TLSv1.2 protocol, in order to protect the information from unauthorized users. In fact, TLSv1.2 allows to generate a shared secret key between the IoTS and each IoTC (Key1, Key2) that could be used to encrypt/decrypt data based on private-key algorithms.

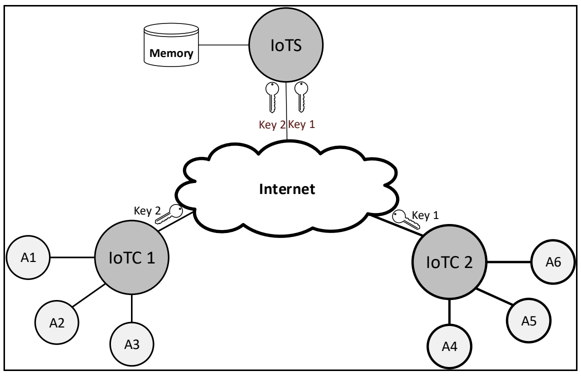


Figure 1. Global Scheme of the targeted IoT (Internet of Things) application.

In this paper, we focus on securing data transferred from/to IoT coordinators by means of the TLS protocol, since SSL is considered insecure [11]. Efficient implementations of these protocols as embedded cryptosystems can be problematic, since the target devices are usually very limited in terms of power, resources and timing. Several TLS/SSL embedded cryptosystem implementations have been proposed in the literature [12–17]. OpenSSL [18,19] is the most deployed library for TLS/SSL applications through software implementations of basic cryptographic functions. For only-software TLS/SSL implementations [17,20], servers can be overloaded with heavy cryptographic operations, which results in long response times. To alleviate this bottleneck, dedicated hardware coprocessors [12–17,21] have been proposed, as Network Security Processors (NSP), as a solution to free these severs from cryptographic operations for flexible management. Nevertheless, although effective efforts have been made [10] for the acceleration of encryption methods, NSPs can provoke an overhead of hardware resources utilization [12,16] to achieve high-performance, due to the required intensive computations within cryptographic algorithms. This constraint paves the way for a HW/SW co-design implementation approach to provide a trade-off between security, area and speed. This approach is based on implementing the computing-intensive cryptosystems in hardware [22,23], while the control of TLS/SSL protocols is performed in software using microprocessors. In this context, Field Programmable Gate Array (FPGA) devices are suitable platforms, as they provide reconfigurability, flexibility and high performance. This is of special interest for the new FPGA generations; such as Zynq from Xilinx or Stratix 10 SoC from Intel, which are equipped with advanced components in a single chip including ARM microprocessors, Advanced eXtensible Interface (AXI) buses, embedded memory or DSPs, and completely match the System on Chip (SoC) paradigm.

In this work, we present a carefully designed SW/HW implementation of the client/server TLSv1.2 protocol for IoTCs and IoTSs, which is implemented on low-cost FPGAs/SoCs suitable for IoT applications. The use of modern FPGA-based SoCs enables the achievement of an optimal trade-off between security, flexibility, area, and speed. Flexibility ensures the possibility of easier algorithm modifications, while leaving the hardware architecture fixed. Among the supported TLS cipher-suites, we have selected Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) [24], Elliptic Curve Digital Signature Algorithm (ECDSA) [25], Advanced Encryption Standard (AES-128) [3], Secure Keyed-hash message authentication (HMAC) [26] and Secure Hash Algorithm (SHA256) for our implementation. These algorithms are all combined to generate 384-bit TLS secret shared keys. The interest on ECCs [7] is justified by the fact that these systems provide better security with smaller key sizes when compared to the RSA method [27], and they are especially suitable for hardware implementation when binary fields are used [28].

Therefore, the paper provides two main contributions: the first one is the proposed SW/HW partitioning for efficient TLSv1.2 negotiations. The main idea is to implement the core operation of ECC, which is ECSM, within a scalable hardware coprocessor accelerator and to integrate it around an ARM microprocessor. Meanwhile, the control of ECDHE and ECDSA protocols, the execution of AES-128 algorithm, HMAC and SHA256 functions are ensured by the ARM microprocessor. The second contribution is the proposed internal architecture of the ECC accelerator, with low area requirements while maintaining high performance. It is based on time-area optimized finite field units and the use of dual-port block RAMs as registers. In addition, the I/Os of this ECC accelerator are 32-bit wide, which allow an easier integration with 32-bit microprocessors (e.g., ARM, PowerPC and Microblaze) via 32-bit buses (e.g., AXI and PLB).

The rest of this paper is organized as follows: Section 2 presents the TLSv1.2 handshake protocol and the considered ECC cryptosystems. Section 3 is devoted to the description of the internal architecture of our ECC accelerator. The proposed FPGA-based IoTS and IoTC designs, the performance evaluation on a Xilinx Zynq device and comparisons with other works in the literature are illustrated in Section 4. Finally, conclusions are presented in Section 5.

2. Transport Layer Security Protocol

The TLSv1.2 protocol allows to generate a shared private key between IoTSs and IoTCs for each session based on cipher suite agreed during the TLS handshake. A demonstration of the TLS handshake between IoTC and IoTS is shown in Figure 2.

The negotiations are based on sending and receiving records, which are blocks of data. Initially, TLS1.2 begins with ClientHello() (step 1), in which the IoTC provides the cipher suite of the supported cryptographic algorithms and compression methods. It also provides random client data ($\text{Rand}_{\text{IoTC}}$) to be used later in the handshake. Then, the IoTS replies with ServerHello() (step 2) by providing random server data ($\text{Rand}_{\text{IoTS}}$) and the list of the selected cryptographic and compression methods to be used during the TLS process. In the proposed designs, TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 is the supported TLS cipher suite. Once the Hello step is done, the IoTS and the IoTC calculate in parallel a pair of private/public ephemeral keypairs (steps 3 and 4) using an EC-based keypair generation algorithm [7] and send to the other party the public key. The server uses ECDSA to sign its ephemeral public key (Ps) in Step 4 and sends the signature to the client. On the other side, the IoTC verifies the received signature using the ECDSA verification algorithm (Step 5). If the verification is successful, the IoTC sends its public key (Pc). Then, a 384-bit shared secret key will be generated (Step 6) by the combination of ECDHE and HMAC-SHA256. The first algorithm provides a 256-bit PreMasterSecret key, while, the second generates a 384-bit MasterSecret key. From the latter, two 128-bit (client_write_key, server_write_key) secret keys are extracted. Finally, in order to check if the handshake was not tampered with (Step 7), the IoTC and IoTS encrypt “ping” and “pong” using the AES algorithm by server write key and client write key, respectively. Then, they exchange the encrypted messages, and each part decrypts the received message using the appropriate key to retrieve "ping" and "pong" messages. Otherwise, the TLS handshake process was tampered.

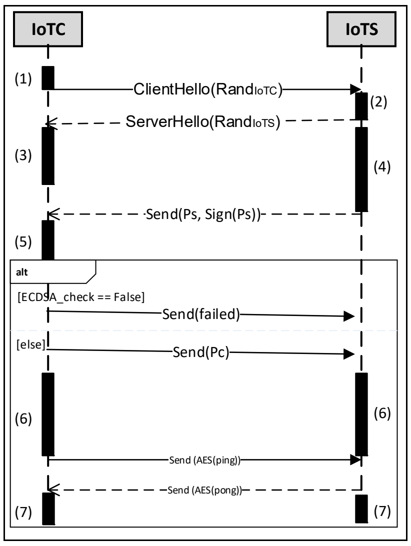


Figure 2. Transport Layer Security (TLS) Handshake demonstration.

2.1. Elliptic Curve Cryptography

In ECC, most of elliptic curves are defined over prime fields (Fp) and binary fields (GF(2^m)) [29]. As binary fields are more suitable for hardware implementations [23], in this paper we are interested in GF(2^m), where the field element k is a binary of fixed length m:

$$k = (k_{m-1}k_{m-2} \ldots k_1k_0) \,/\, k_i \in \mathrm{GF}(2) \tag{1}$$

The arithmetic is defined by the polynomial representation:

$$K(t) = (k_{m-1}{\cdot}t^{m-1} + k_{m-2}{\cdot}t^{m-2} + k_1{\cdot}t + k_0) \tag{2}$$

An Elliptic Curve E defined over GF(2^m) consists of a set of points P represented by the coordinates (x_p,y_p), where x_p and y_p are elements of GF(2^m) solving the Weierstrass expression [28]:

$$y^2 + x{\cdot}y = x^3 + a{\cdot}x^2 + b \,/\, (a,b) \in \mathrm{GF}(2^m) \tag{3}$$

The conception of ECC schemes consists of three parts, namely, the curve parameters domain, the key generation and the encryption/decryption algorithms. In the literature, several standard curve domains are recommended with different key-length, m, where m is prime number in the set {163, 233, 239, 283, 409, 571}. The *sect233r1* (NIST B-233) curve [30], defined over GF(2^{233}), is widely used in TLS1.2. It is recommended for HW implementations when high speed and less area consumption are intended [31].

The EC-based key generation algorithm results in the (d,Q) keypair, where the private key d is an integer of m-bits and Q is a point on E. The keypair (d,Q) is calculated as follows:

- Choose an integer d from $[1, 2^m - 1]$.
- Calculate $Q = d \times G$, where G is the generator point defined by sect233r1.

In the literature, several standard cryptographic protocols based on ECs are reported. In this work, we will use ECDHE_ECDSA protocols, as they are used in TLS1.2 for secret key exchange and digital signature, respectively. In the following, we present the ECDHE and ECDSA algorithms.

2.1.1. Elliptic Curve Diffie Hellman Ephemeral

ECDH stands for EC-based Diffie-Hellman key agreement protocol. It ensures the establishment of a secret shared key between two parties through an insecure channel. This key could be used by a symmetric cryptosystem for data encryption. In the literature, two versions of ECDH are reported, namely, ECDH static and ECDH Ephemeral (ECDHE). The difference is that the first version always uses the same keypairs, while the second generates new keypairs for each connection. The ephemeral version is recommended in TLS protocols. The shared secret key is obtained by applying the following steps:

- IoTC chooses integer $n1$ from $[1, 2^m - 1]$ and computes $Q1 = n1 \times G$.
- In parallel, IoTS chooses integer $n2$ from $[1, 2^m - 1]$ and computes $Q2 = n2 \times G$.
- IoTC and IoTS exchange $Q1$ and $Q2$.
- IoTC and IoTS compute $Q = n1 \times Q2$ and $Q = n2 \times Q1$, respectively.
- Extract the shared secret key from the coordinates of the shared point Q.

We note that ECDHE requires the execution of four ECSMs. However, the computation of $Q1$ and $Q2$ are performed in parallel, as well as the computation of $Q = n1 \times Q2$ and $Q = n2 \times Q1$. The execution time (T_{ECDHE}) of ECDHE algorithm can be estimated as:

$$T_{ECDHE} \sim T_{rand} + 2 \cdot T_{ECSM} \tag{4}$$

where T_{rand} represents the execution time of the m-bit secure random generation and T_{ECSM} corresponds to the execution time of single ECSM.

2.1.2. Elliptic Curve Digital Signature Algorithm

ECDSA is an EC-based DSA algorithm proposed in 1992 by Scott Vanstone [25]. It is used for data integrity to avoid message tampering during transfer by signing the message. This protocol consists of two algorithms, namely signature generation and signature verification. In our work, the first procedure is performed by the IoTS to sign its ephemeral public key. Meanwhile, the second procedure is executed by the IoTC to check if the received public key is appropriate to the server or to a third-part. Pseudocode descriptions of the two algorithms are presented in Algorithm 1 and Algorithm 2, respectively, while their detailed justification and description can be found in [32,33].

Algorithm 1. Elliptic Curve Digital Signature Generation.

```
Inputs: private key $d$, message $msg$, domain parameters $(m, a, b, G, n, h)$
Outputs: Signature $(r, s)$

1. Generate random integer $k \in [1, n - 1]$
2. Compute $e = \text{Hash}(msg)$
3. Compute $R = k \times G$
4. Set $r = x_R \bmod n$. If $r = 0$ return to step1
5. Compute $s = (k^{-1} \times (e + d \times r)) \bmod n$
6. The signature for $msg$ is then $(r, s)$
```

In our work, the message msg of Algorithm 1 is the concatenation of the coordinate of the IoTS ephemeral public key. The resulting signature of the message msg is represented by (r, s). The execution time (T_{alg1}) of Algorithm 1 can be estimated as:

$$T_{alg1} \sim T_{rand} + T_{Hash} + T_{ECSM} + T_{MI} + 2 \cdot T_{MM} + T_{MA} \tag{5}$$

T_{alg1} is linked to the following execution times: secure random generation of k (T_{rand}), secure hash function (T_{Hash}) for e computation and single ECSM (T_{ECSM}) for computing the coordinates of the point R, Modular Inversion (T_{MI}), two Modular Multiplications (T_{MM}) and Modular Addition (T_{MA}) to obtain s.

According to Algorithm 2, the verification of the signature requires the execution of the secure hash function, the computation of a MI and two MMs for v, $u1$ and $u2$ calculations.

Algorithm 2. Elliptic Curve Digital Signature Verification.

```
Inputs: message $msg$, signature $(r, s)$, domain parameters $(m, a, b, G, n, h)$, senders public key $P$
Outputs: accept or reject signature
1. verify $r, s \in [1, n - 1]$
2. compute $e = \text{Hash}(msg)$
3. compute $v = s^{-1} \bmod n$
4. compute $u1 = e \times v \bmod n$
5. compute $u2 = r \times v \bmod n$
6. compute $X = u1 \times G + u2 \times P$
7. if $X = O \rightarrow$ reject signature
8. else if $X_x \bmod n = r \rightarrow$ accept signature
```

To compute the coordinates of the point X, two ECSMs and single Elliptic Curve Point Addition (ECPA) are required. The execution time (T_{alg2}) of Algorithm 2 could be estimated by Equation (6).

$$T_{alg2} \sim T_{Hash} + 2 \cdot T_{ECSM} + T_{ECPA} + T_{MI} + 2 \cdot T_{MM}, \tag{6}$$

To use ECDHE_ECDSA, both the IoTC and IoTS are required to be able to perform ECSM, which is the main operation of most ECC protocols. This operation is considered as the most expensive operation for embedded systems in terms of hardware requirements and timing performance. Therefore, we propose to implement a dedicated ECC hardware accelerator for high-speed ECSM computation in order to enhance the overall performance of TLS execution, while also taking in consideration the area usage. In the following, the considered ECSM algorithm and the internal hardware architecture of ECC coprocessor are described.

3. ECC Accelerator Design

Depending on the representation of the scalar and the points, several fast and regular ECSM algorithms are reported in the literature [34]. In this work, the ECSM is performed based on the Montgomery Power Ladder (MPL) algorithm over projective coordinate system [29]. Making field operations explicit, this algorithm uses the binary representation of the scalar k as it is shown in Algorithm 3. The use of the MPL algorithm [35] is often suggested to withstand side channel attacks by performing the Elliptic Curve Point Addition (ECPA) and Elliptic Curve Point Doubling (ECPD) in parallel regardless of the current scalar bit value. In the other hand, the introduction of the projective point coordinate system [36] within ECPA and ECPD computations ensures high performance by avoiding Modular Inversion (MI) execution at each iteration of the main loop. This operation is the most complex and costly to implement on embedded systems when compared to Modular Addition (MA), Modular Squaring (MS) and Modular Multiplication (MM), also required in ECPA and ECPD calculations. Hence, the combination of MPL and projective system allows the enhancement not only of design security but also of the overall cryptosystem performance. These features enable the proposed algorithms for efficient ECC hardware implementations when security, high-speed and low-area requirements are targeted.

The computation of ECSM based on Algorithm 3 requires three steps: initialization (lines 1 and 2), main loop (lines 3 to 11) and calculation of the resulting point coordinates (lines 12 and 13). The first step performs two field squarings (line 1) and a single ECPD (line 2). The second step performs, at each iteration, ECPA (lines 5 and 8) followed by ECPD (lines 6 and 9). These operations are executed in the projective system by performing a set of field additions, field multiplications and a field squaring. In the final step, two field inversions are required (lines 12 and 13) to obtain the coordinates (x_3,y_3) of the resulting point.

Algorithm 3. Montgomery ladder over projective coordinates, making field operations explicit.

```
Inputs: $k=k_{m-1}k_{m-2}\ldots k_1k_0$, $P(x,y)$, domain parameters (m, $a$, $b$, $G$, $n$, $h$)
Outputs: $k \times P = (x_3, y_3)$
1. $X_1 = x$, $Z_1 = 1$, $X_2 = x^4 + b$, $Z_2 = x^2$
2. $P_1 = P$, $P_2 = 2P$ \\ ECPD
3. for $i = m - 2$ down to 0 do
4. If ($k_I$ == 0) then
5. $T = Z_2$; $Z_2 = (X_1 \times T + X_2 Z_1)^2$; $X_2 = x Z_2 + X_1 X_2 Z_1 T$ \\ ECPA
6. $T = X_1$; $X_1 = T^4 + b Z_1^4$; $Z_1 = T^2 Z_1^2$ \\ ECPD
7. else
8. $T = Z_1$; $Z_1 = (X_1Z_2 + X_2 T)^2$; $X_1 = xZ_1 + X_1X_2Z_2T$ \\ ECPA
9. $T = X_2$; $X_2 = T^4 + bZ_2^4$; $Z_2 = T^2Z_2^2$ \\ ECPD
10. end if
11. end for
12. $x_3 = X_1Z_1^{(-1)}$
13. $y_3 = (x + x_3)\,[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{(-1)} + y$
```

The internal architecture of the proposed ECC accelerator for ECSM computation based on Algorithm 3 is presented in Figure 3.

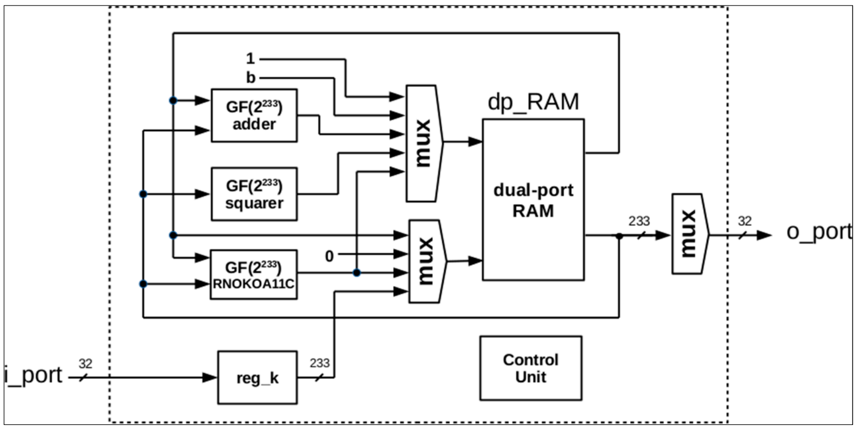


Figure 3. Hardware architecture of ECC (Elliptic Curve Cryptosystems) accelerator.

The proposed architecture consists of three finite field arithmetic units over GF(2^{233}) (an adder, a squarer and a multiplier, called RNOKOA11C), dual-port RAM (dp_RAM), a 233-bit register (reg_k), a control unit, and three multiplexers (mux). The field units ensure the computations of field addition, field squaring and field multiplication, respectively. The dp_RAM block is used for storing the coordinates (x,y) of the point P, the intermediate results of Algorithm 3 and the coordinates (x_3,y_3) of the resulting point. Meanwhile, the reg_k register is used to store the scalar value k to manage the main loop. It is also useful as temporary storage of (x,y) values before transferring them to the RAM. The control unit is responsible for the coordination between the integrated components of the internal architecture for performing ECSM. The proposed architecture provides an excellent trade-off between area and performance, based on the following aspects:

1. Exploiting the block RAMs available in FPGA devices within the internal architecture of ECC accelerator instead of using registers, thus saving Look-Up Tables (LUT) resources at the expense of introducing some extra clock cycles.
2. Integrating the I/O interface into the ECC processing unit, taking advantage of the displacement reg_k.
3. Avoiding the use of a dedicated field divider/inverter, by means of using the Itoh-Tsujii algorithm (ITA) [7], thus requiring only the multiplier and the squaring units. In this case, our ECC accelerator needs 353 clock cycles for performing 231 squarings and 10 multiplications required for GF(2^{233}) field inversion execution. This performance overhead is assumable, taking into account that only two field inversions are required.

It is worth mentioning, that the input (i_port) and the output (o_port) ports of the proposed ECC accelerator are 32-bit wide. It means that this ECC accelerator can be easily integrated around various 32-bit microcontroller through 32-bit buses.

3.1. Field Multiplier Unit

As shown in Algorithm 3, the field multiplier is the one having the most noticeable effect on the performance of the scalar-point multiplying, thus requiring a careful design. The RNOKOA11C multiplier unit is implemented based on an improvement of the Karatsuba–Ofman Algorithm (KOA) [37], named Non-Overlapping KOA (NOKOA) multiplier [38]. The NOKOA multiplier allows to perform field multiplication in only one clock cycle, thus enabling high-performance ECC accelerators. However, area requirements are excessive for its implementation on low-cost devices [22]. In [22], two modifications of NOKOA, requiring 3 and 9 clock cycles for completing a field multiplication, are presented. These modifications, named NOKOA3C and NOKOA9C, respectively, require less area but are not suitable for use in our ECC scalar-point multiplication unit, due to the lack of output registers. In fact, the use of RAM blocks instead of registers makes necessary to register the result provided by the multiplier. Figure 4 shows the proposed architecture of the RNOKOA11C multiplier, which meets the requirements imposed by the used of RAM as registers. It presents a recursive structure, thus consisting on a lower-level NOKOA multiplier, a control unit, two multiplexers, two XOR networks, and the RT, RE, RO and MO registers. This new multiplier requires 11 clock cycles for performing a field multiplication

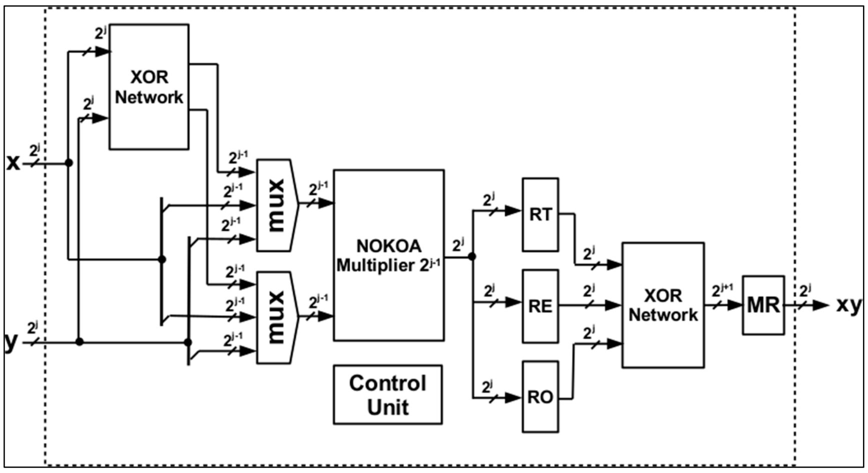


Figure 4. Internal architecture of RNOKOA11C field multiplier unit.

Table 1 shows synthesis results comparing NOKOA9C [22] to RNOKOA11C multipliers over GF(2^{233}) finite field. These results have been obtained using Xilinx ISE 14.4 over Virtex 5 devices (xv5vlx110-3f1760). As it is shown, the number of LUTs is almost the same, because the additional register required by RNOKOA11C is included into the LUTs occupied by the XOR network. Small differences in the number of LUTs and delay are due to optimizations performed by the software tool. Regarding the number of clock cycles, RNOKOA11C requires 11 clock cycles instead of the 9 clock cycles required by NOKOA9C, but it fits the requirements for being the multiplier unit of our ECC accelerator, which has been named MP_ECC_B-233_RNOKOA11C.

Table 1. Synthesis results for NOKOA9C and RNOKOA11C over GF(2^{233}) on Virtex 5 devices.

Design	# LUTS	# Max. Freq. (MHz)	# Cycles	Total Time @50MHz	Total Time @Max. Freq.
NOKOA9C	2366	214	9	0.18 µs	42 ns
RNOKOA11C	2344	205	11	0.22 µs	54 ns

3.2. Implementation of MP ECC_B-233_RNOKOA11C

In order to check the suitability of MP_ECC_B-233_RNOKOA11C for medium-performance applications, such as IoT coordinators/gateways, it has been implemented in a MiniZed board [39] with a Zynq 7Z007S device from Xilinx. This low-cost device includes a single-core ARM Cortex-A9 microprocessor and 14400 LUTs of programmable logic for software/hardware co-design. The software tool used for this implementation has been Vivado 2018.2 from Xilinx. Also, for comparison purposes, it has been implemented on Virtex 5 devices using Xilinx ISE 14.4. Implementation results are presented in Table 2, where MP_ECC_B-233_NOKOA11C is compared to other ECC scalar-point multipliers with similar area.

Table 2. MP_ECC_B-233_RNOKOA11C implementation results and comparison to other implementations.

Device	Design	# LUTS	# Cycles	Time @50 MHz
Virtex 4 (xc4vlx200)	Ansari [40]	13396	5890	117 µs
Virtex 7 (xc7v585_T)	Khan [41]	7895	5924	118 µs
Virtex 5 (xc5vlx110-3)	Sutter [42]	13244	8193	163 µs
ZynQ (xc7z020-1)	Parrilla [22] (NOKOA9C)	6223	14013	315 µs
Virtex 5 (xc5vlx110-3)	This work (RNOKOA11C)	3203	20637	413 µs
ZynQ (xc7Z007s)	This work (RNOKOA11C)	3395	20637	413 µs

From Table 2, it is evident that this new design requires less than half the area of other implementations, while providing similar performance figures. Thus, it is perfectly suitable for the target application. It should also be noted that our design includes a 32-bit I/O interface, while the other alternatives do not include such feature.

4. FPGA Implementation of TLS Cryptosystem

Among the considered TLS cipher-suites, HMAC, SHA256 and AES are characterized by its high-performance implementation due to a relative mathematic simplicity. ECDHE and ECDSA are characterized by its high security but are considered the most time/area consuming as they involve complex operations over large prime numbers. To achieve the best trade-off between flexibility, area and speed, a SW/HW co-design implementation approach is presented in this work. The proposed partitioning is based on the implementation of ECSM within a compact ECC hardware accelerator for faster execution. The dedicated core is integrated around an embedded ARM microprocessor. The rest of the required operations for TLS negotiation are managed in SW by the processor. Figure 5 presents the hardware architecture of the proposed embedded system. The hardware architecture was implemented on the Xilinx Zynq-7Z007S SoC device in the Avnet Minized Dev board [39] for both IoTS and IoTC coordinators. As commented in the previous section, this low-cost device consists of a single-core ARM Cortex-A9 microprocessor, able to run at up to 666.666 MHZ, along with 100 block RAMs and 14400 slice LUTs for software/hardware co-design. The MiniZed board also includes a Murata "Type 1DX" LBEE5KL1DX wireless module for wireless communications.

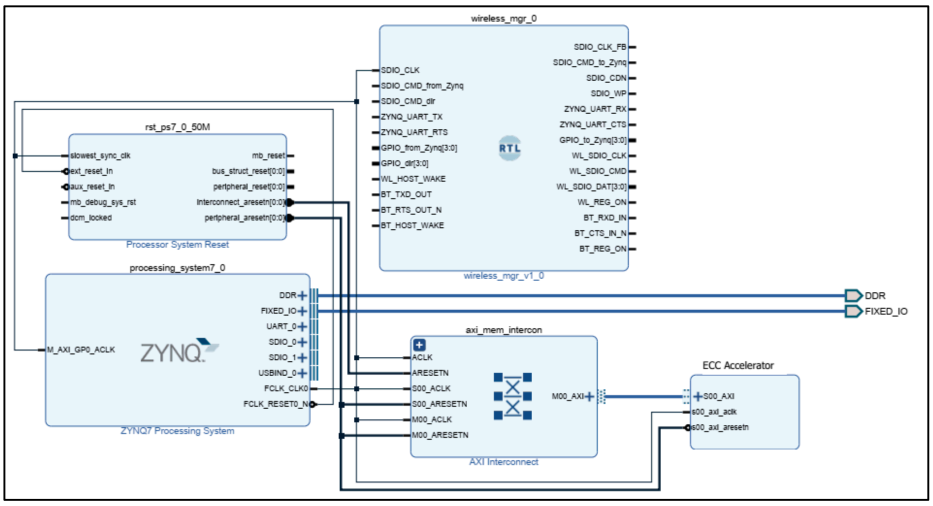


Figure 5. Hardware architecture of IoTS (IoT Server coordinator) and IoTC (IoT Client coordinator) designs.

The proposed architecture contains a single Cortex-A9 ARM microprocessor (PS), the MP_ECC_B-233_RNOKOA11C accelerator, an AXI interconnect bus and a Wireless_mgr controller. The latter is used for the WiFi connection of the IoTS and IoTC designs with gateways that provide internet access. The AXI bus allows 32-bit data/instruction exchanges between the ARM microprocessor and the ECC accelerator. It runs with a 50 MHz clock. The ARM processor ensures not only the control of the ECC accelerator but also of all TLS processes. The roles assigned to the processor are defined as follows:

- Generation of 256-bit random numbers.
- Execution of AES, HMAC and SHA256 functions.
- Computation of finite field inversions, multiplications and additions required for ECDSA.
- Control of the MP_ECC_B-233_RNOKOA11C accelerator.
- Control of ECDHE and ECDSA algorithms.
- Control of internet communication between the IoTS and the IoTCs.

4.1. ECC Accelerator Integration around ARM Processor

For connecting the ECC accelerator with the ARM processor through the AXI bus, Xilinx's Intellectual Property InterFace (IPIF) is used for 32-bit data/instruction exchanging, as it is shown in Figure 6. The IPIF is configured with four 32-bit registers: InsIn, DataIn, InsOut and DataOut. The processor uses a set of instruction codes through the InsIn register to manage the ECC core. The second register is used to transfer the digits of the input point coordinates and the scalar from the ARM to the req_k register. The control unit makes use of the third register to notify the processor that the coordinates of the resulting point from the ECSM computations are ready. The last register ensures the transfer of the resulting point coordinates to the processor.

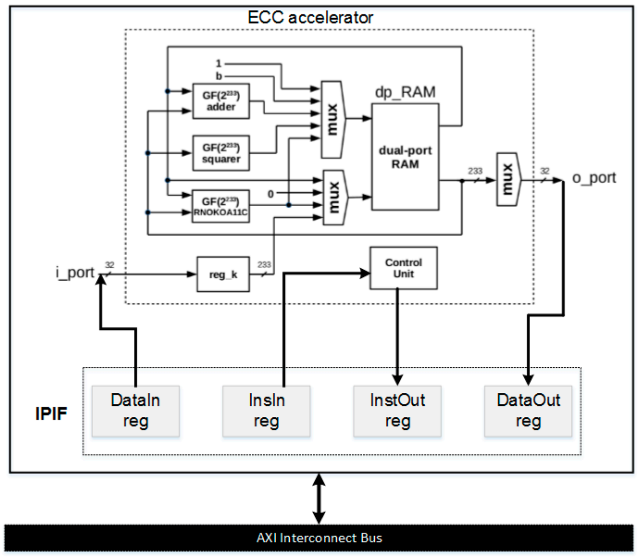


Figure 6. Integration of ECC accelerator with AXI (Advanced eXtensible Interface) bus.

To perform ECSM, three steps are required for each execution, namely, ECC core reset, transmission of the inputs, and retrieving of the resulting point coordinates. Before starting the ECSM computation, the ARM processor resets the ECC accelerator by sending the 0 x 000000001 instruction. After that, the control unit stores sixteen 8-bit digits of the ECSM input point coordinates followed by eight 8-bit digits of the scalar transmitted from the processor to the dp_RAM. It must be noted that the processor transmits the 0x000000002 instruction after each digit to prepare the control unit to receive the next digit. Once twenty-four 8-bit digits of the inputs are loaded, the control unit manages the field units to perform ECSM computations. During this time, the InsOut register value is 0 x 000000000. When the ECC accelerator completes the execution, the control unit changes the InsOut register value to 0 x 000000003 in order to notify the processor that the ECSM execution is done, then sends sixteen 8-bit digits of the resulting point coordinates. The processor uses the 0 x 000000004 instruction after receiving each digit to order the control unit to send the next digit.

Table 3 summarizes the hardware resources occupied by the ECC accelerator and the proposed architecture for IoTS and IoTC coordinators on the Zynq-7Z007S device. The results are shown in terms of slice LUTs and selected RAM blocks.

Table 3. Hardware resources requirements of the proposed architectures.

Design	# LUTS	RAMs
ECC accelerator	3395	7
IoTS	8503	9
IoTC	8503	9

From Table 3, it must be noted that the difference in hardware resources between IoT designs and the ECC accelerator is 5108 LUTs and 2 RAMs. This is due to the AXI interconnect bus and the wireless_mgr controller. The proposed ECC accelerator requires only 24% of the total available LUTs in the targeted device. Meanwhile, the overall design occupies 60% of them. Moreover, the proposed architecture requires only 9 block RAMs.

4.2. Software Development

The proposed IoTS and IoTC coordinators run on Embedded Linux by loading the Linux boot image for Zynq (BOOT.bin) and the Linux system image (image.ub) files to the QSPI flash and the eMMC memory, respectively, both available on the board. These files are generated by means of Xilinx

Petalinux 2018.2 tool based on the hardware description file (bitstream.bit) of the proposed hardware architecture. The idea behind the use of embedded Linux is that the OS allows flexible use of the WiFi module for internet communication between the IoTS and IoTCs using TCP/IP client/server sockets. Figure 7 summarizes the software development required to implement the TLS1.2 protocol.

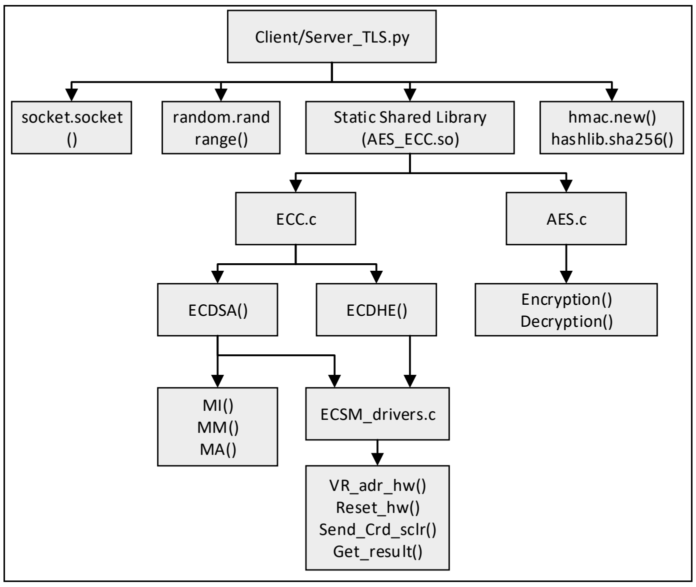

Client/Server_TLS.py
socket.socket ()
random.rand range()
Static Shared Library (AES_ECC.so)
hmac.new() hashlib.sha256()
ECC.c
AES.c
ECDSA()
ECDHE()
Encryption() Decryption()
MI() MM() MA()
ECSM_drivers.c
VR_adr_hw() Reset_hw() Send_Crd_sclr() Get_result()

Figure 7. TLS software functions.

To implement this TLS1.2 protocol between the IoTS and IoTC coordinators, Server_TLS.py and Client_TLS.py python codes have been developed for each design, respectively. Python has been used in order to exploit socket, random, hashlib and hmac libraries for TCP/IP socket communication, random generation, SHA256 and HMAC executions, respectively. Since Python is interpreted code, which makes its execution slower, we propose to implement the AES and ECC algorithms in C for faster executions. Then, we generate the static shared library (AES_ECC.so) from the resulting C code to be imported and used in Client/Server_TLS.py files. The C code and the static shared library are generated using the Xilinx Software Development Kit (XDSK) tool. The shared library consists of two C function files, namely AES.c and ECC.c. The first file defines AES encryption() and decryption() functions. The second file describes ECDSA() and ECDHE() functions for performing the considered ECC protocols. The two functions are based on the ECC_driver.c file and finite field functions required in the ECDSA algorithm. It must be noted that the inputs and the outputs of the AES and ECC functions are based on radix-2^8 and radix-2^{32} representations, respectively. Radix-2^8 is used since the AES algorithm performs 8-bit operations, while, radix-2^{32} is considered for ECC algorithms not only because the ARM is a 32-bit microprocessor but also for the AXI 32-bit bus where data/instruction are transferred digit-by-digit in serial mode. The representation of large numbers in radix 2^8 and radix 2^{32} is performed in Python based on the ctypes.c_int library. The ECDSA() function requires the computation of MA, MM and MI. In the ECDSA protocol, MA, MM and MI computations over 256-bit operands are required for Algorithm 1 and Algorithm 2. These computations are ensured by the MA(), MM() and MI() functions. In fact, MM is performed based on Montgomery radix-2^{32} Modular Multiplication algorithm [27,43]. In the other hand, MI is executed by modular exponentiation (Mexp) according to Fermat's little theorem [7], as it is shown in Equation (7). This theorem and, consequently, Equation (7), are valid when n is prime integer. The easiest way to perform Mexp is the left-to-right binary method [44].

$$A^{-1} \bmod n = A^{n-2} \bmod n \tag{7}$$

The ECC_driver.c file contains C drivers to control the ECC accelerator. Itis composed of four functions: reset_hw(), send_crd_sclr(), Get_result() and VR_adr_hw(). The first three functions allow to reset the ECC accelerator, send the inputs of Algorithm 3, and retrieve the resulting point coordinates, respectively. As our designs run on embedded Linux, the ARM processor needs at system initialization to generate a virtual address (ECC_vr_adr) for the ECC accelerator and map it to its physical address (ECC_BASE_ADDR). This step is ensured by the VR_adr_hw() function, where the following instructions are executed:

1. int fd = open("/dev/mem",O_RDWR);
2. int pg_size = sysconf(_SC_PAGESIZE);
3. int pg_adr_ECC = ECC_BASE_ADDR & (pg_size-1);
4. int pg_offset_ECC = ECC_BASE_ADDR - pg_adr_ECC;
5. ECC_vr_adr = mmap(NULL, pg_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, (ECC_BASE_ADDR & (pg_size-1)));

Once the virtual address is generated, the addresses of the four registers used for data/instruction exchanging can be calculated as follows:

- InsIn_adr = *((unsigned *)(ECC_vr_adr+pg_offset_ECC))
- DataIn_adr = *((unsigned *)(ECC_vr_adr+pg_offset_ECC+4))
- InsOut_adr = *((unsigned *)(ECC_vr_adr+pg_offset_ECC+8))
- DataOut_adr = *((unsigned *)(ECC_vr_adr+pg_offset_ECC+12))

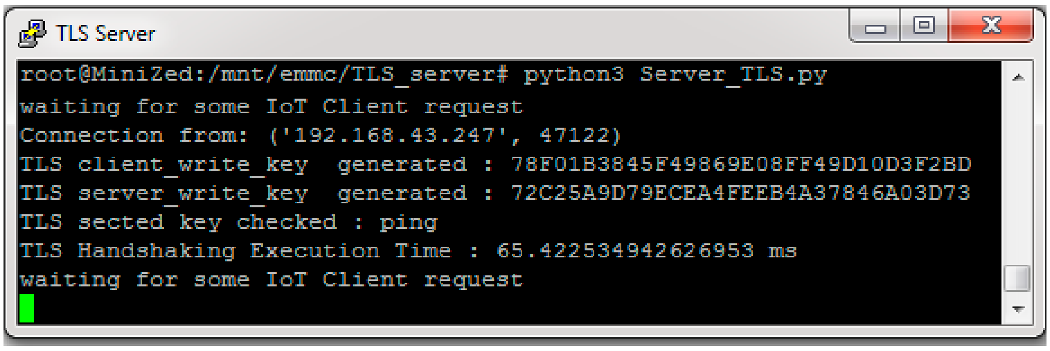
Table 4 presents the execution time of the developed crypto functions for the TLS1.2 protocol, as well as the time of all of the TLS1.2 process. The reported performances include the following execution times:

- Random generation.
- SHA256 and HMAC functions.
- Data representation from large numbers to radix-r for AES and ECC computations.
- Client/Server data exchanging via sockets.

Table 4. Execution time of the involved crypto functions for TLS execution.

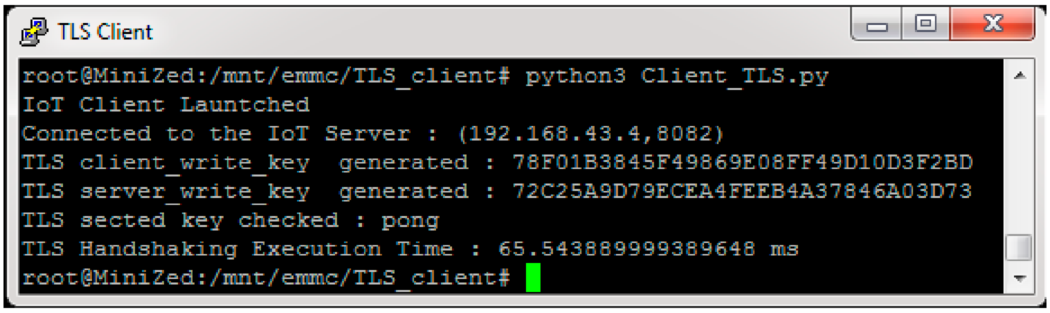
Protocol	Bit-Length	Function	Execution Time
AES	128 bits	AES_encryption()	56 µs
		AES_decryption()	101 µs
ECC	233 bits	ECSM()	413 µs
		ECDHE()	1.7 ms
		ECDSA_gen()	3.5 ms
		ECDSA_check()	4.1 ms
TLS	384 bits	TLS1.2()	67.5 ms

The proposed design performs a single 233-bit ECSM using the ECC accelerator in 400 µs. Moreover, the IoTS and IoTC perform the ECDHE procedure in 1.7 ms. This time depends on the size of $n1$ and $n2$, which are required in the ECDHE procedure. In our case, the size of both $n1$ and $n2$ is 233 bits. For the ECDSA protocol, the IoTS generates the signature in 3.5 ms, while the IoTC checks the received signature in 4.1 ms. These two times are linked to the bit-size of the k generated in line 1 of Algorithm 1 and the intermediate results ($u1$, $u2$) of Algorithm 2. Finally, the generation of the 384-bit secret key between the IoTS and the IoTC based on the TLS1.2 protocol is achieved in 67.5 ms. Figure 8 shows a screenshot of one measurement of TLS1.2() execution in the server side (Figure 8a), and the client side (Figure 8b).



```
root@MiniZed:/mnt/emmc/TLS_server# python3 Server_TLS.py
waiting for some IoT Client request
Connection from: ('192.168.43.247', 47122)
TLS client_write_key  generated : 78F01B3845F49869E08FF49D10D3F2BD
TLS server_write_key  generated : 72C25A9D79ECEA4FEEB4A37846A03D73
TLS sected key checked : ping
TLS Handshaking Execution Time : 65.422534942626953 ms
waiting for some IoT Client request
```

(**a**)



```
root@MiniZed:/mnt/emmc/TLS_client# python3 Client_TLS.py
IoT Client Launtched
Connected to the IoT Server : (192.168.43.4,8082)
TLS client_write_key  generated : 78F01B3845F49869E08FF49D10D3F2BD
TLS server_write_key  generated : 72C25A9D79ECEA4FEEB4A37846A03D73
TLS sected key checked : pong
TLS Handshaking Execution Time : 65.543889999389648 ms
root@MiniZed:/mnt/emmc/TLS_client#
```

(**b**)

Figure 8. Screenshot of TLS1.2() execution time in the (**a**) server side, and the (**b**) client side.

4.3. Comparison with Some Recent Works

In order to compare our proposal with other works, an ad hoc experimental setup have been prepared, which consists of two Minized boards hosting Zynq devices and communicated using WiFI. Both boards' devices include the MP_ECC_B-233_RNOKOA11C accelerator for cryptographic operations, while one the Minizad boards acts as server and the other one acts as a client for the TLS handshaking. Figure 9 shows a picture of this experimental setup.

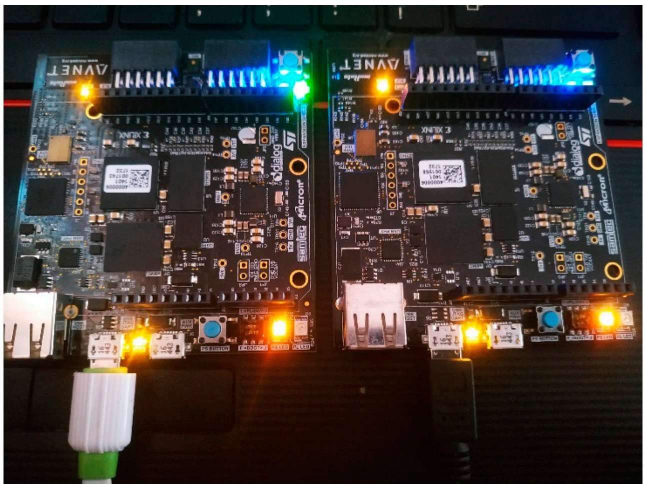


Figure 9. Experimental setup for TLS handshaking.

Table 5 shows the performance comparison of our design and some FPGA-based TLS/SSL implementations. The comparisons are made in terms of occupied slice LUTs, selected RAM blocks and execution time for single TLS/SSL handshake negotiations.

Table 5. TLSv1.2 implementation performance comparisons to recent works.

Device	Design	Approach	Freq MHz	# LUTs	RAMs	Execution Time
Zynq-7Z007S	This work	SW/HW	666/100	8503	9	67.5 ms
Spartan-3	Wang [16]	HW	150	90644	216	0.62 ms
Virtex 5	Hamilton [12]	HW	75	39052	75	11.3 ms
Zynq- 7z020	Paul [14]	SW/HW	- /125	27559	-	-
Huawei-Taushia	Xiao [17]	SW/HW	2100/-	-	-	59241 kB/s (*)
Virtex 7	Paul [21]	HW	-	52005	225	220 ms

(*) the authors report the performance only in terms of throughput.

The authors of [16] presented a Network Security Processor (NSP) implementation on a Spartan-3 FPGA device of the IPSec/SSL protocols. The results show that their processor provides high timing performance by achieving 1600 full SSL handshakes per second with a 150 MHz clock. However, it requires 10 times more slice LUTs and 24 times more RAMs than our design.

In [12], a FPGA-based NSP of the TLSv1.2 protocol on a Virtex-5 device was proposed. The NSP was implemented with a secure true random number generator and ECC coprocessor. Compared to our design, the proposed processor is 6 times faster. However, it requires 5 times more slice LUTs and 9 times more RAMs.

In [14] a pipelined architecture of an NSP for the SSL/TLS protocols is implemented on a Zynq-7z020-clg484 device. The proposed NSP presents high area requirements with 3 times more slice LUTs than our design. The authors did not present the timing performance of the TLS/SSL handshake.

We note that these implementations [12,14,16] present high-speed processors but with high-area requirements. Hence, these designs are not recommended for low-area FPGA devices, as opposed to the contrary of our design, which can be efficiently used on such devices.

In [17], a SW/HW implementation of an Energy-Efficient Crypto Accelerator (EECA) for an HTTPS server on a 8-Core HUAWEI Taishan server and an ARM Cortex-A57 CPU was proposed. The evaluation of the whole Web server was reported in terms of throughput and energy consumption for different data sizes, ranging from 1 KB to 2 MB. The obtained throughputs vary from 59241 KB/s to 1001 KB/s with a 2.1 GHz clock. The high-performance of this HTTPS server is obtained by using very expensive hardware platforms, as once more opposed to our implementation targeting low-cost FPGA devices.

The authors of [21] presented the implementation of the TLSv1.3 protocol for end-to-end secure connection between an Intel i5 client trusted workplace and a Virtex-7 FPGA cloud node (SecFPGA). The proposed design takes about 220 ms to perform the TLSv1.3 handshake and to deploy a 4 MB file. It requires 52005 LUTs and 225 RAMs. Thus, our design shows better time execution while requiring less area.

5. Conclusions

In this paper, FPGA-based Client/Server designs, implemented on a Zynq FPGA device, of the TLSv1.2 protocol for IoT applications are presented. Our main aim is to achieve the best trade-off between flexibility, scalability, timing execution, and area consumption, with special attention to area requirements for enabling low-cost IoT implementations while maintaining good performance figures. To improve the execution time, a SW/HW co-design implementation approach is proposed. Thus, the critical point operation ECSM of ECC protocols is implemented in HW around an ARM Cortex A9 microprocessor, while, the control of the whole TLSv1.2 handshake negotiations is ensured by the processor, which runs on embedded Linux OS for Zynq. The proposed 32-bit I/O ECC accelerator requires only 3395 slice LUTs, thus allowing not only flexible integration around various 32-bit microprocessors but also an easier implementation on low-cost FPGA devices. The proposed architecture occupies 8503 LUTs and performs full handshake negotiations between IoTS and IoTC designs in 67.5 ms. From the performance comparisons of our results and other works in the literature, it can be concluded that our design achieves the best trade-off between security, area and speed for the target application. It requires less area while providing reduced timing execution. Therefore, the proposed implementation approach is suitable for small IoT embedded Client/Server secure coordinators implemented on low-cost devices.

Author Contributions: Conceptualization, L.P.; Methodology, J.A.Á-B., E.C. and L.P.; Software, A.M.B. and J.A.Á-B.; Validation, L.P. and A.G.; Formal Analysis, A.M.B. and L.P.; Resources, L.P.; Writing—Original Draft Preparation, A.M.B. and L.P.; Writing—Review & Editing, L.P. and A.G.; Supervision, L.P., N.B. and M.A.; Project Administration, L.P., N.B. and M.A.

Funding: This work was partially supported by Ministry of Higher Education and Scientific Research of Algeria under scholarship program "Exceptional National Program (PNE) 2018-2019".

Conflicts of Interest: The authors declare no conflict of interest

References

1. Wang, S.; Hou, Y.; Gao, F.; Ji, X. A novel IoT access architecture for vehicle monitoring system. In Proceedings of the 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), Reston, VA, USA, 12–14 December 2016.
2. Dierks, T.; Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.2*, Internet Engineering Task Force, IETF, RFC 5246 (Proposed Standard), Updated by RFCs 5746, 5878, 6176T. 2008.
3. NIST. *Advanced Encryption Standard (AES) (FIPS–197)*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2001.
4. NIST. *Data Encryption Standard (DES) (FIPS–46-3)*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 1999.
5. NIST. *Secure Hash Standard (SHS) (FIPS 180-4)*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2015.
6. NIST. *Secure Hash Standard (SHS) (FIPS 202)*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2015.
7. Hankerson, D.; Menezes, A.J.; Vanstone, S. *Guide to Elliptic Curve Cryptography*; Springer: Berlin/Heidelberg, Germany, 2003; p. 332.
8. Dofe, J.; Frey, J.; Yu, Q. Hardware security assurance in emerging IoT applications. In Proceedings of the 2016 IEEE International Symposium on Circuits and Systems (ISCAS), Montreal, QC, Canada, 22–25 May 2016.
9. Tao, H.; Bhuiyan, M.Z.A.; Abdalla, A.N.; Hassan, M.M.; Zain, J.M.; Hayajneh, T. Secured Data Collection with Hardware-Based Ciphers for IoT-Based Healthcare. *IEEE Internet Things J.* **2019**, *6*, 410–420. [CrossRef]
10. Al-Omary, A.; Alsabbagh, H.M.; Al-Rizzo, H. Survey of Hardware-based Security support for IoT/CPS Systems. *KnE Eng.* **2018**, *3*, 52–70. [CrossRef]
11. Moeller, B.D.T.; Ko towicz, K. *This POODLE Bites: Exploiting the SSL 3.0 Fallback*. Security Advisory. 2014. Available online: https://www.openssl.org/~{}bodo/ssl-poodle.pdf (accessed on 28 October 2019).
12. Hamilton, M.; Marnane, W.P. Implementation of a secure TLS coprocessor on an FPGA. *Microprocess. Microsyst.* **2016**, *40*, 167–180. [CrossRef]
13. Khalil-Hani, M.; Nambiar, V.P.; Marsono, M.N. Hardware Acceleration of OpenSSL Cryptographic Functions for High-Performance Internet Security. In Proceedings of the 2010 International Conference on Intelligent Systems, Modelling and Simulation, Liverpool, UK, 27–29 January 2010.
14. Paul, R.; Chakrabarti, A.; Ghosh, R. Multi core SSL/TLS security processor architecture and its FPGA prototype design with automated preferential algorithm. *Microprocess. Microsyst.* **2016**, *40*, 124–136. [CrossRef]
15. Paul, R.; Shukla, S. Partitioned security processor architecture on FPGA platform. *IET Comput. Digit. Tech.* **2018**, *12*, 216–226. [CrossRef]
16. Wang, H.; Bai, G.; Chen, H. A Gbps IPSec SSL Security Processor Design and Implementation in an FPGA Prototyping Platform. *J. Signal Process Syst.* **2010**, *58*, 311–324. [CrossRef]
17. Xiao, C.; Zhang, L.; Liu, W.; Bergmann, N.; Xie, Y. Energy-efficient crypto acceleration with HW/SW co-design for HTTPS. *Future Gener. Comput. Syst.* **2019**, *96*, 336–347. [CrossRef]
18. Roy, D.B.; Agrawal, S.; Reberio, C.; Mukhopadhyay, D. Accelerating OpenSSL's ECC with low cost reconfigurable hardware. In Proceedings of the 2016 International Symposium on Integrated Circuits (ISIC), Singapore, 12–14 December 2016.

19. Viega, J.; Chandra, P.; Messier, M. *Network Security with Openssl*; O'Reilly & Associates, Inc.: Sebastopol, CA, USA, 2002; p. 384.
20. Wu, L.; Weaver, C.; Austin, T. CryptoManiac: A fast flexible architecture for secure communication. In Proceedings of the 28th Annual International Symposium on Computer Architecture, Gothenburg, Sweden, 30 June–4 July 2001.
21. Genssler, P.R.; Knodel, O.; Spallek, R.G. Securing Virtualized FPGAs for an Untrusted Cloud. In Proceedings of the ESCS'18, Las Vegas, NV, USA, 30 July–2 August 2018.
22. Parrilla, L.; Álvarez-Bermejo, J.A.; Castillo, E.; López-Ramos, J.A.; Morales-Santos, D.P.; García, A. Elliptic Curve Cryptography hardware accelerator for high-performance secure servers. *J. Supercomput.* **2019**, *75*, 1107–1122. [CrossRef]
23. Parrilla, L.; Castillo, E.; López-Ramos, J.A.; Álvarez-Bermejo, J.A.; García, A.; Morales, D.P. Unified Compact ECC-AES Co-Processor with Group-Key Support for IoT Devices in Wireless Sensor Networks. *Sensors* **2018**, *18*, 251. [CrossRef]
24. Blake-Wilson, S.; Bolyard, N.; Gupta, V.; Hawk, C.; Moeller, B. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*; RFC 4492; Internet Engineering Task Force (IETF), 2006; Available online: https://tools.ietf.org/html/rfc4492 (accessed on 28 October 2019).
25. Johnson, D.; Menezes, A.; Vanstone, S. The Elliptic Curve Digital Signature Algorithm (ECDSA). *Int. J. Inf. Secur.* **2001**, *1*, 36–63. [CrossRef]
26. Bellare, M.; Canetti, R.; Krawczyk, H. *Keying Hash Functions for Message Authentication*; Advances in Cryptology — CRYPTO '96. CRYPTO 1996. Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1996; Volume 1109.
27. Bellemou, A.; Benblidia, N.; Anane, M.; Issad, M. MicroBlaze-Based Multiprocessor embedded cryptosystem on FPGA for Elliptic Curve Scalar Multiplication over Fp. *J. Circuits Syst. Comput.* **2018**, *28*, 1950037. [CrossRef]
28. Koblitz, N. Elliptic curve cryptosystems. *Math. Comput.* **1987**, *48*, 109–203. [CrossRef]
29. Cohen, H.; Frey, G.; Avanzi, R.; Doche, C.; Lange, T.; Nguyen, K.; Vercauteren, F. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, 2nd ed.; Chapman & Hall/CRC: Boca Raton, FL, USA, 2012; p. 1024.
30. Certicom Research. *SEC 2: Recommended Elliptic Curve Domain Parametes*, Version 2.0, Standards for Efficient Cryptography; 2010. Available online: https://www.secg.org/sec1-v2.pdf (accessed on 28 October 2019).
31. Huang, L.; Adhikarla, S.; Boneh, D.; Jackson, C. An Experimental Study of TLS Forward Secrecy Deployments. *IEEE Internet Comput.* **2014**, *18*, 43–51. [CrossRef]
32. IEEE. *IEEE Standard Specifications for Public-Key Cryptography*; IEEE Std 1363-2000; IEEE: Piscataway, NJ, USA, 2000; ISBN 978-0-7381-1957-1.
33. IEEE. *IEEE Standard Specifications for Public-Key Cryptography—Amendment 1: Additional Techniques*; IEEE Std 1363a-2004; IEEE: Piscataway, NJ, USA, 2004; ISBN 978-0-7381-4004-9.
34. Rivain, M. *Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves*. IACR Cryptology ePrint Archive; Report 2011/388 2011. Available online: https://eprint.iacr.org/2011/388 (accessed on 28 October 2019).
35. Joye, M.; Yen, S.-M. *The Montgomery Powering Ladder*; Springer: Berlin/Heidelberg, Germany, 2003.
36. Baldwin, B.; Goundar, R.R.; Hamilton, M.; Marnane, W.P. Co-Z ECC scalar multiplications for hardware, software and hardware–software co-design on embedded systems. *J. Cryptogr. Eng.* **2012**, *2*, 221–240. [CrossRef]
37. Karatsuba, A. Math The complexity of computations. *Proc. Steklov Inst. Math.* **1995**, *211*, 169–183.
38. Fan, H.; Sun, J.; Gu, M.; Lam, K.-Y. Overlap-free Karatsuba-Ofman polynomial multiplication algorithms. *IET Inf. Secur.* **2010**, *4*, 8–14. [CrossRef]
39. Avnet. *Minized Board Datasheet*. Available online: http://zedboard.org/sites/default/files/documentations/MiniZed-GSG-v1_2.pdf (accessed on 28 October 2019).
40. Ansari, B.; Hasan, M.A. High-Performance Architecture of Elliptic Curve Scalar Multiplication. *IEEE Trans. Comput.* **2008**, *57*, 1443–1453. [CrossRef]
41. Khan, Z.; Benaissa, M. Throughput/Area-efficient ECC Processor Using Montgomery Point Multiplication on FPGA. *IEEE Trans. Circuits Syst. II Express Briefs* **2015**, *62*, 1078–1082. [CrossRef]
42. Sutter, G.; Deschamps, J.; Imaña, J. Efficient Elliptic Curve Point Multiplication using Digit Serial Binary Field Operations. *IEEE Trans. Ind. Electron.* **2013**, *60*, 217–225. [CrossRef]

43. Issad, M.; Boudraa, B.; Anane, M.; Bellemou, A.M. Efficient PSoC Implementation of Modular Multiplication and Exponentiation Based on Serial-Parallel Combination. *J. Circuits Syst. Comput.* **2019**. [CrossRef]
44. Issad, M.; Boudraa, B.; Anane, M.; Anane, N. Software/Hardware Co-Design of Modular Exponentiation for Efficient Rsa Cryptosystem. *J. Circuits Syst. Comput.* **2014**, *23*, 1450032. [CrossRef]